Generics and Collections

13

LEARNING OBJECTIVES

After completing this chapter, you will be able to understand:

- Generic programming and its application in Java.
- How generics can be employed to improve program functionality and provide room for improvement in the programs.
- Object collections and the functionality that they can offer in Java.
- Important APIs and Java methods that help you use object collections for data manipulation.

13.1 | Introduction

Java is a powerful programming language that facilitates programmers by combining innovative object-oriented functions with legacy options. In this chapter, we focus on a largely ignored part of programming languages, which is the ability to deal with abstract algorithm structures that are later capable of providing the required level of support. Object collections also allow programmers to create and manipulate sets of objects that may offer extended functionality. We will discuss the generics and object collections that can be employed in Java.

This chapter presents a thorough discussion of the implementation classes, application program interface (API), and other options that are available for use in the language, such as mapping and collecting functions. This chapter stresses on the methods that describe the use of generics as well as the important collection schemes that are perfect for use with Java.

We will first establish the objectives of the chapter, then give brief descriptions of the terms of generics and collections in software engineering, and finally discuss the specific functions that you can employ in Java object collections and generic programming methods.

We aim to ensure that this chapter provides the required information for improving your generic programming applications and ensure that you take full advantage of an OOP language that provides object collections for use in complex programming applications.

13.2 | Generic Programming

Generic programming is a key concept in software engineering. It involves the use of algorithms that have the ability to create data types, which can be later specified by instantiating them with a particular set of object or type parameters. This approach has a respectable history and is designed to ensure that it is possible to provide functionality, which can reduce duplication in the program code. Multiple languages employ this functionality, including Java.

Generic programming concept is termed as parametric polymorphism in some programming languages, because it allows the language to have elements and code blocks that can have different forms based on the requirements of a program. Abstract program code is perfect for use in conditions where different programming patterns need to be employed from effectively the same form of code. Generics are also great to ensure that programming errors can be reduced by forcing programmers to use safer methods that allow them to stay away from logical errors.

Java employs generics as a programming facility, which is present in the type system identification in the language. It can be implemented as an object or method type, which is designed to cover operations on different data types, while ensuring that compiling errors are eliminated and a safety net is implemented that takes away logical errors.

Generics in Java have a strong structure which contains type parameters like the following. Variables provide the parameters, which ensures a structure of the class.

```
class class_name<T1, T2, ..., Tn> {}
```

A generic class is capable of declaring multiple type variables, such as Integer and String, in angle brackets (<>). See the following example:

```
class MyClass <Integer, String, Boolean> {}
```

The above code ensures that the class has the required set of parameter types such as Integer, String, and Boolean, which can cover any possible invocation of the class parameters during application.

A generic interface is one that contains type variables, which act as parameters and provide the same interface during the runtime phase of program execution. A generic method is similar to normal Java method, and it can type parameters that may either be following a class object or a Java interface. A generic constructor can work independently in Java apart from its class. This is possible because the Java collections framework is capable of dealing with constructors separately, which may have a parameter list of a generic class or the interface.

13.2.1 Benefits of Generics

Generics have their specific benefits, which suggest that programmers need to use them over other coding schemes. When employed with a programming language such as Java, it allows classes and interfaces to act as parameters and therefore, set up parameters that have greater usability. It is possible to use different inputs and reuse the same code to a greater benefit, which is the main appeal of employing generics.

Java code having generics contains other benefits. Let us take the example of employing data type checks, which occur at the time of compilation. Generics ensure that the compiler can pick up on any errors and generate an error code that describes the missing type safety. This problem is easy to identify and resolve. The same problem goes into the runtime phase if not generics is not used. It becomes a time-consuming exercise to detect, debug, and remove the logical errors that are present in the employed datatypes.

Another benefit of using generics is that it eliminates the use of casting. It is common for the compiler to cast datatypes when adding items in array lists. With the use of generics, the problem is simply eliminated as the required functionality is embedded within the program. The following is an example when casting does not need to be explicitly mentioned:

```
package java11.fundamentals.chapter13;

import java.util.ArrayList;
import java.util.List;

public class Casting {
    public static void main(String args[]){
        List<String> mylist = new ArrayList<String>();
        mylist.add("I love Arraylist");
        String mystring = mylist.get(0);
        System.out.println(mystring);
    }
}
```

The above program produces the following output.

```
I love Arraylist
```

This use of generics ensures that there is no need to perform type casting and change items between primitive data types and the Wrapper classes in Java. Another key benefit of using generics is that it allows Java programmers to develop good habits

and then employ generic frameworks which are type safe, easier to read, and can be customized as and when required during a specific coding application. The main benefits of using generics is to have flexibility in your programs, while ensuring type safety of the objects and methods that you employ.

You can set up structures in your programs that offer improved usability as a complete package. Since you do not specify the fixed data types in your interfaces and generic classes, you can come up with a code that can work with different collections of datasets, such as ones that require a combination of integer and string values. You can avoid the problems that you will face when using non-generics and get a superior code, especially with the use of a single interface definition that offers a strong type control over the employed data objects.

13.2.2 Using Generics in Java

There are various generic elements that you can employ in Java. It is possible to appreciate the benefits of Java if you can compare it with the use of a traditional code to handle the same task. Let us take the example of the Box class for this purpose, where we will describe how it is possible to create a generic version of the class, ensuring that we get excellent benefits when we go ahead with programming and expanding on the created base class.

A simple class will employ private objects where we will have to use two methods of `set()` and `get()` to ensure that the methods employed can use the class object. This does not mean that we can control how the object will be employed during the compilation phase, as it is possible to place a String when the object type is set to work as an integer object in the program code implementation.

This problem is easily resolved when we create a generic version of the same class. A generic version is created using the angle brackets "< >". It ensures that the type declaration that we use employs a variable that we can protect. The following is the code example that shows the use of a generic class, which will offer enhanced type safety:

```
public class Box<T> {
    // where T describes the Type of the object use
    private T t;
    public void set(T t) { this.t = t; }
    public T get() { return t; }
}
```

Here, we can clearly observe that the use of the type "T" that we have set up in the generic class declaration replaces all instances of the use of an object in the code. This ensures that the type variable can follow any array, class, another type variable, or any other non-primitive value providing better control and enhanced functionality in the Box class.

There are some conventions that you must follow when employing generic programming. The type parameters are always single capital letters. As it is difficult to use any name without setting up conventions, you may find it difficult to understand generic programs created by other developers. The common letters used by programmers often employ specific parameters that denote the following values:

1. **E:** It denotes an element and you will find it employed by the Java Collections Framework, which we will define in subsequent sections.
2. **K:** It denotes key and will be employed for pointing to various data objects and displaying the object information accordingly.
3. **N:** It stands for number value, where we aim to employ them as parameters for the generic code.
4. **T:** It denotes type. We can employ a particular type that we want to use in the program and this will offer type safety, which is a characteristic feature of generics in Java.
5. **V:** It describes value. We can employ the type accordingly when we implement this in a generic class or object.
6. **S, U, etc.:** These denote that we are employing multiple types. The letters will be denoting them successively, such as S for 2nd and U for 3rd types.

Once you have set up a general class, you need to implement an invocation which needs to present a concrete value. This can include an instance such as that of integer, which is an object. You pass the type argument to your generic class, which is different from using an argument with a class method. This locks up the generics and ensures that it is not possible to employ wrong arguments. Remember, the code will not generate a new object, rather, it will employ the use of a parameter and will produce a reference to your original generic class, which will be of the Integer type in your example:

```
Box <Integer> intBox;
```

Once you have instantiated a generic type in Java, you can continue to work with it in the program, using different type interfaces as well as employ various parameters. We have already described that it is possible to have any number of type parameters in a generic class. This allows us to use objects that may belong to various Wrapper classes, as a common use is to implement functionality for both Integer and String objects.

One way to create a generic interface is to set up an ordered pair of parameters. This can then be set to have keys and values as parameters, allowing the use of the programs in a variety of ways. The use of a generic interface is easier as you do not have to worry as to whether you are using types or objects, as autoboxing in Java will always ensure that it is possible to pass data types to a class, which are automatically converted according to the required use.

It is also possible to use parametrized types that are present within the confines of a generic class definition. Creating such a type is possible in the following manner, if we continue with our earlier example:

```
Box<Integer> intBox = new Box<>();
```

Now, this creates a raw type. This use of code may not be generic in some older versions, but Java fully supports the use of API classes and other detailed generic functionality, which is included in the compiler functionality.

13.2.3 Generic Methods

Now that we have explained the use of generic classes, especially the ones that work with parametrized types, it is time that we describe the generic methods. Creating a generic method is similar to creating a generic type, but the advantage here is that the scope of this type remains limited to a method. It is possible to set up both static and non-static generic methods, while they can also be implemented in the form of class constructions.

If we take the example of a static generic method, we find that the type parameter must be defined before the return type of the method is mentioned in the code. An example for invoking a method for this purpose may be in the form:

```
Pair<String, Integer>pa1 = new Pair<>("grape", 3);
```

It is also possible to use type inference which allows the use of a generic method as a typical Java method. This means that there is no need to specify the type of parameters in the angle brackets.

Now, we describe the benefit of using a generic method, which is to bound the parameters and ensure only restricted types that can be employed as method arguments. This can be achieved by creating the upper bound for the type and use the "extends" keyword as shown in the following example, like `<U extends Number>`.

```
public class Box<T> {
    private T t;
    public void set(T t) {
        this.t = t;
    }
    public T get() {
        return t;
    }
    public <U extends Number> void inspect(U u){
        System.out.println("T: " + t.getClass().getName());
        System.out.println("U: " + u.getClass().getName());
    }
    public static void main(String[] args) {
        Box<Integer> integerBox = new Box<Integer>();
        integerBox.set(new Integer(15));
        integerBox.inspect("some text"); // error: is presented since this would still be
returning a String
    }
}
```

This is an excellent example of using a method which works with a locked number type. Since we have inserted some string in the inspect method, this directly translates to a String object. This means that the compiler finds that the method cannot work with the available information, since the parameter describes a number extension limiting the parameters that the `integerBox` can employ. It is also possible to use further complications such as multiple bounds, but we are limiting our use here to the examples that we have presented.

An important concept to understand is that Java is an object-oriented programming (OOP) language. An important principle of OOP is that of inheritance. Inheritance allows creating relationships between different objects and classes. The same case is possible when using generics, where we can set up relations between different elements of generic code.

Creating subtypes of generic classes and interfaces is possible by either extending the class or implementing the interface. The relationship that we create between the various types of classes is possible by using clauses to define it. Here is an example:

```
interface MyList<E,T> extends List<E>{
}
```

This is a simple example where `MyList` is a generic interface which extends the `List` with a single parameter value. Here, it is possible to see that a generic list capable of accepting a string object can have a subtype which can accept an Integer object as well as a String object. There are wildcards as well that can be employed when setting up generic code. "?" is the most common wildcard and it denotes a currently unknown type. It can be used for a local variable, a field, a parameter value, and a return type.

Generics in Java are designed to check for errors at the compile time. It does not have a use at the time of runtime. The compiler has the function of "type erasure" as an important feature, which eliminates all the type checking code from the final program when creating the byte code. It may cast any type if it is required. The parameterized types do not create any new classes, and therefore, generics in Java do not have any additional overhead runtime resource use.

There are some things that are locked in Generics, such as creating subtypes at a fundamental unit. It is also not possible to create Generic arrays as they will not compile. The idea behind using generics is to create and store specific objects in your Java program. Since the type erasure occurs before the execution phase, the parameters cannot be designed to set up during the runtime phase of the program. This may restrict the programming approach, although it is possible to set up higher parameters such as objects, which still allow the use of subunits such as Integer or String.

Java Generics are constantly on the rise and developed in its advanced forms. Java 10, which came out in March 2018, has the experimental Project Valhalla present in it. It contains several generics improvements, such as the use of specialized lists and reified generics, which ensure that the actual types can be made available at runtime, even when employing generic code.

Before we can explore the world of Collections, it is important to understand two most important methods that are needed for collections to work properly. These methods are `hashCode()`, and `equals()`. In the following section, we will see why these methods are important and why one should override them.

13.2.4 Overriding `toString()`, `hashCode()`, and `equals()`

Everything in Java other than primitives is an object. Anything you can think of like exception, events, or arrays extends from `java.lang.Object`. Hence, everything contains the methods shown in Table 13.1 that reside in `java.lang.Object`.

Table 13.1 Methods from `java.lang.Object`

Method	Description
`boolean equals (Object obj)`	This method is used to check if two given objects are meaningfully equivalent
`void finalize()`	This method is used by the garbage collector when it determines that there are no more references to the object
`int hashCode()`	Hashtable, HashMap, and HashSet Collection classes use hashing; this method returns hashcode int value for an object
`final void notify()`	This method is used to wake up a thread which is waiting for the object's lock on which it is used
`final void notifyAll ()`	This method is used to wake up all the threads that are waiting for the object's lock on which it is used
`final void wait()`	This method is used to make current thread to wait till other thread class `notify()` or `notifyAll()`
`String toString()`	This returns the text representation of an object

13.2.4.1 Overriding `toString()` Method

This method is useful in giving some meaningful information about the object. If you do not override this method, the object's default `toString()` will be called, which does not give any meaningful information. It only returns the `classname` followed by @ symbol and the object's hashcode. Let us see the following example:

```
package java11.fundamentals.chapter13;
public class Car {
    public static void main(String args[]) {
        Car car = new Car();
        System.out.println(car.toString());
    }
}
```

The above code gives the following result:

```
java11.fundamentals.chapter13.Car@6e8dacdf
```

Similarly, let us see another example of `toString()` in a side-by-side comparison view.

Without `toString()`

```
class Student{
 int rollno;
 String name;
 String city;

 Student(int rollno, String name,
String city){
 this.rollno=rollno;
 this.name=name;
 this.city=city;
 }

 public static void main(String args[]){
Student s1=new Student(101,"Raj","lucknow");
Student s2=new Student(102,"Vijay","ghaziabad");

 System.out.println(s1);//compiler writes
here s1.toString()
 System.out.println(s2);//compiler writes
here s2.toString()
 }
}
```

```
Output:Student@1fee6fc
Student@1eed786
```

With `toString()`

```
class Student{
 int rollno;
 String name;
 String city;

 Student(int rollno, String name,
String city){
 this.rollno=rollno;
 this.name=name;
 this.city=city;
 }

 public String toString(){//overriding
the toString() method
  return rollno+" "+name+" "+city;
 }
 public static void main(String args[]){
   Student s1=new Student(101,"Raj","lucknow");
   Student s2=new Student(102,"Vijay",
"ghaziabad");

   System.out.println(s1);//
compiler writes here s1.toString()
   System.out.println(s2);//
compiler writes here s2.toString()
 }
}
```

```
Output:101 Raj lucknow
102 Vijay ghaziabad
```

In the overridden method, you can give some meaningful information about the object, which describes the object for better understanding. For example, if you have a customer class, you can give customer related information, such as name and account number ,so when someone calls `toString()` on that object, it will give useful information about that customer.

```
package java11.fundamentals.chapter13;
public class ToStringExampleWithOverriddenToString {
    public static void main(String args[]) {
        ToStringExampleWithOverriddenToString tse = new ToStringExampleWithOverriddenTo-
String();
        System.out.println(tse.toString());
    }
    public String toString() {
        return "This is an overridden method";
    }
}
```

The above code gives the following result.

```
This is an overridden method
```

As you can see, it prints the text which we have provided in the `toString()` method.

13.2.4.2 Overriding `equals()` Method

As you have seen, a simple way to compare two variables is to use ==. However, == is going to check if two references are pointing to the same object or not because == simply looks at the bits in the variable, and checks if they are equal or not. If you only care about checking if two object references are equal, feel free to use ==. However, this is not the case most of the times.

In case of objects, we always want to make sure they are meaningfully equal. Now, == can only tell us if these two references pointing to the same object on the heap or not. Hence, we need to override `equals()` method and compare the necessary attributes of the class, which will tell us if they are truly equals. It is also important to make sure which variables you use for the comparisons. If you do not use unique ones per object, you are going to get the wrong result. For example, in case of a Car class, two objects of Car may have similar attributes, such as color, make, and model. If you are using these attributes in the `equals()` method to compare, you may end up concluding that two completely different cars are same, even though they are owned by different persons. Hence, in this case, you may want to compare on a unique element of that class, such as registration number. This will give you the correct result for the comparison.

13.2.4.2.1 Importance of Overriding `equals()` Method

For collections, it is important to override class's `equals()` method because without overriding and giving meaningful equal comparison, it is not possible to find the object in the collections. Hence, we will not be able to use the object of this class as a key in a hashtable or HashMap. Let us explore this in detail. If we do not override `equals()` method, then the Object's `equals()` method will be used instead as everything extends from Object. This Object's `equals()` method uses only == operator for comparisons. This == comparison means two objects are treated equal if those two references refer to the same object.

Now, let us look at an example where we will not be overriding the `equals()` method. Let us assume that we have two houses that are being sold at the government registry office. Your program is to put house object and owner object in a HashMap, where the house object is the key and owner is the value. Now, when we want to retrieve the information after finishing the process, we need to provide the house object to get the owner's information. This is tricky as we now need to provide the exact reference to the object we used as the key when we added it to the collection. Remember, we cannot create an identical house object again and use it for search. It is because this newly created object will refer to a different location on the Heap, and since Object's `equals() method is` using == to compare two objects, it is only comparing the objects' memory locations to check if they are same. Even though these two objects are logically equal as per the == comparison, they are different. Hence, we will never be able to create identical object and we will fail to get the data out of HashMap. This is why we will not be able to use an object as a hashtable key or HashMap key without overriding `equals()` method and comparing two objects for logical equivalence.

How do we solve this problem with overriding `equals()` method? We need to make sure we use the suitable attribute to compare two objects so that they can be meaningfully equivalent despite referring to two different objects on the Heap. So, in

our example, let us assume that we can use house address as a comparison attribute in the `equals()` method. Now, we can compare two addresses to make sure these two houses objects are same and hence we can retrieve the owner object from the HashMap. However, we can still improve on this a little. Instead of using house object as the key, we can simply use house address as the key so that we will not need to create an identical object reference in the future to retrieve the owner object. We can simply use address String to get the owner object from the HashMap. This is possible as String and Wrapper classes work well as keys in hashtables, HashMaps, etc., as they override the `equals()` method.

```
package java11.fundamentals.chapter13;
public class EqualsMethodTest {
    public static void main(String[] args) {
        Car myCar = new Car("Mercedez Benz","S Class", "MH05 12345");
        Car carInGarage = new Car("Mercedez Benz","S Class", "MH05 12345");
        if (myCar.equals(carInGarage)) {
            System.out.println("Yay!!! This is my Car!");
        }
    }
}
class Car {
    private String brand;
    private String model;
    private String registrationNumber;
    Car(String brand, String model, String registrationNumber) {
        this.brand = brand;
        this.model = model;
        this.registrationNumber = registrationNumber;
    }
    public boolean equals(Object o) {
        if (o instanceof Car) {
            Car car = (Car) o;
            if (car.getBrand() == this.brand && car.getModel() == this.getModel() && car.
getRegistrationNumber() == this.getRegistrationNumber()) {
                return true;
            } else {
                return false;
            }
        } else {
            return false;
        }
    }

    public String getBrand() {
        return brand;
    }
    public void setBrand(String brand) {
        this.brand = brand;
    }
    public String getModel() {
        return model;
    }
    public void setModel(String model) {
        this.model = model;
    }
    public String getRegistrationNumber() {
        return registrationNumber;
    }
    public void setRegistrationNumber(String registrationNumber) {
        this.registrationNumber = registrationNumber;
    }
}
```

The above code produces the following result.

```
Yay!!! This is my Car!
```

In the above example, class Car overrides `equals()` method which compares car brands, models and registration numbers using == operator. Since we are comparing Strings, we can safely use == operator for comparison. String is immutable which means string's content will not change once created. And any string objects that we create are stored in the String Constant Pool. This pool cannot have two objects with same content. Hence, if we create another object with the same content, it will only point to the same object in the heap instead of creating a new reference. Therefore, == on two strings will give proper result.

13.2.4.2.2 The `equals()` Method Contract

Every class overriding `equals()` method must adhere to the `equals()` contract mentioned below, which is taken from the Java Docs:

1. **Reflexive:** For any reference value `x`, `x.equals(x)` should return true.
2. **Symmetric:** For any reference values `x` and `y`, `x.equals(y)` should return true if and only if `y.equals(x)` returns true.
3. **Transitive:** For any reference values `x`, `y`, and `z`, if `x.equals(y)` returns true and `y.equals(z)` returns true, then `x.equals(z)` must return true.
4. **Consistent:** For any reference values `x` and `y`, multiple invocations of `x.equals(y)` consistently return true or consistently return false, provided no information used in equals comparisons on the object is modified.
5. For any non-null reference value `x`, `x.equals(null)` should return false.

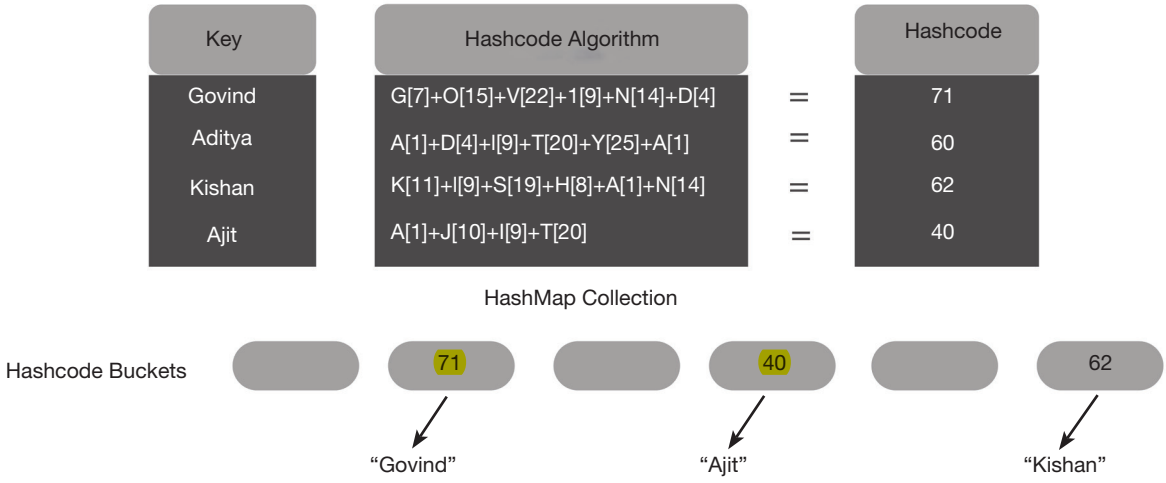
Next, we will look into `hashCode()` method, as `equals()` and `hashCode()` are bound together by a joint contract. This contract states, "if two objects are considered equal using the `equals()` method, they must have identical hashcode values".

Hence, we need to look into overriding `hashCode()` as well to make sure we create truly meaningful equal objects.

13.2.4.3 Overriding `hashCode()`

In collections, you can think of hashcode as an object ID but not necessarily unique, which is used to increase the performance of large collections of data. Collections like HashMap and HashSet use the hashcode of an object to see how to store it in the collection and for its retrieval later. Hence, hashcode plays an important role in collections. We need to make sure we have as unique a hashcode as possible to increase the performance of the collections.

Let us try to understand this with the help of an example. In a room, there are 1000 buckets placed, with labels from 1 to 1000. You are given a bunch of slips. Each slip has a name on it and you need to determine a numeric value of each name by adding the letters' numeric equivalents like A=1, B=2, C=3, etc. Each name will give you a number. You need to place this name slip in the bucket that has this number label on it.



Now, if someone comes with a name and asks to get a slip for that name, we can quickly get the number for corresponding to that name and look for the slip in the bucket. This is how collections work. While creating this hashcode generator algorithm, we need to make sure we generate the same hashcode for the same input. So, each time we pass the same name we get the same number back.

As you may have noticed, we may have more than one number for many names that share same letters, such as Ayaan and Nayaan. Since they share the same letters, they will have same number if we add all the characters. It means one bucket may have multiple slips in it. This is acceptable however, if we make our algorithm efficient and we have distributed load in the buckets. This will also speed up the search operation.

Search is a two-step process. First finding the right bucket using `hashCode()` and then searching the bucket for the right element using `equals()` method.

Now, consider the case where our hashcode generator algorithm does not give us consistent result, and we may get different hashcodes at different times by passing the same name. This could be a problem, as when we first added a name in these hash collections, the name would have gone to a bucket with the name's hashcode value. Now, when we try to retrieve the same name, this hashcode generator algorithm gives different hashcode even though the name is same. In this case, we are not going to find the bucket as our name, when we first added is in completely different bucket than we are looking into. Because of this hashcode generator's inconsistent behavior, we will find no-match even though the objects are equal. Hence, it is important to note that for two objects to consider equal, their hashcodes must also be equal. This is because if the hashcodes were different, the equal test would fail due to scanning into a wrong bucket.

Let us look at the following example, which shows how to override `hashCode()` method.

```
package java11.fundamentals.chapter13;
public class HashCodeTest {
    public int myVar;
    HashCodeTest(int myVar) {
        this.myVar = myVar;
    }
    public boolean equals(Object o) {
        //We need to check if the object is instanceof of HashCodeTest class. If not we
can safely return false
        if(o instanceof HashCodeTest) {
            HashCodeTest hTest = (HashCodeTest) o;
            if (hTest.myVar == this.myVar) {
                return true;
            } else {
                return false;
            }
        }else {
            return false;
        }
    }
    public int hashCode() {
        return (myVar * 23); //You can use your own logic. We tried to multiply by a
prime number.
    }
}
```

13.2.4.3.1 The `hashCode()` Contract

The following contract is taken from the Java API documentation as is. This describes the hashCode contract:

1. Whenever it is invoked on the same object more than once during an execution of a Java application, the `hashCode()` method must consistently return the same integer, provided no information used in `equals()` comparisons on the object is modified. This integer need not remain consistent from one execution of an application to another execution of the same application.
2. If two objects are equal according to the `equals(object)` method, then calling the `hashCode()` method on each of the two objects must produce the same integer result.

3. It is NOT required that if two objects are unequal according to the `equals(Java.lang.Object)` method, then calling the `hashCode()` method on each of the two objects must produce distinct integer results. However, the programmer should be aware that producing distinct integer results for unequal objects may improve the performance of hashtables.

Now, let us try to interpret this contract (Table 13.2).

Table 13.2 `hashCode()` contract

Condition	Required	Allowed but Not Required
`x.equals(y) == true`	`x.hashCode() == y.hashCode()`	
`x.hashCode() == y.hashcode()`		`x.equals(y) == true`
`x.equals(y) == false`		No `hashCode()` requirements
`x.hashCode() != y.hashCode()`	`x.equals(y) == false`	

13.3 | Collections

Object collections are important in Java. They are available as a framework and provide the architecture required to store a collection of objects. The framework and the supporting APIs allow programmers to not only store the collections but carry out various information manipulations. This ensures that it is possible to perform functions, such as String changes and modifications. Figure 13.1 shows Arrays and Collections relationship with Object.

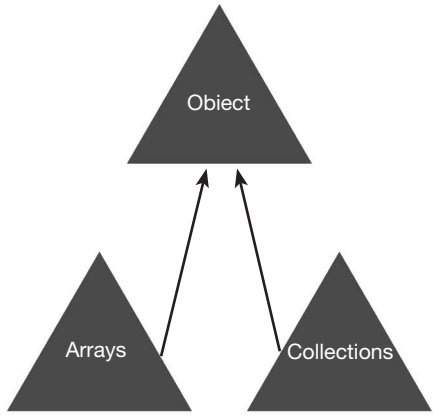


Figure 13.1 Arrays and Collections extends Object.

The use of collections is always optional and allows you to employ the set of objects as a single coherent structure. Usually, the framework contains interfaces and classes to set up the object groups and provide the required functionality to the programmers. This functionality is accessible through the use of collection APIs.

A key point that arises is that arrays also hold primitive objects in Java and therefore, can also be considered as object collections. They are also managed as a singular group item. Thus, it is possible to sort, modify, and work on the data items that are present in an array, as it effectively works as a pointer to the data objects. However, collections are different because their capacity is not assigned when they are instantiated by a programmer.

Collections do not hold the elements of the primitive types. Rather, they hold complete objects belonging to the Primitive Wrapper classes, including Long, Integer, and Double. The earliest Java platform did not have collections implementations, and object manipulation operations had to be carried out using various array options. These were difficult to control and would produce problems during interfacing to have standardized members.

13.3.1 Collections in Java

A concurrency package was developed, which could handle the Java collections. The java.util.Collection contains all the collections implementation facilities that we now have available in JDK and IDE software tools for program development. Java architecture combines collections with program generics and offers three types of collections.

1. The first type is the use of ordered lists. These are important when you want to insert objects in a specific order, such as creating a waiting list.
2. The second type is in the form of a dictionary/map that provides a set of information, which can be searched using a reference key to obtain the value of a particular object.
3. The third type is a set. Sets are simply unordered collections. This means that you cannot have similar objects because there is no way to have distinct objects without having an order.

Each of these collection elements has its specific advantages. The ordered lists are excellent as a more flexible way to implement an array. They provide a storage space for their elements that have a better order. Duplication is possible as long as each element is properly placed in its specific position. All positions are available for search, and the listing can be done by either using linking or creating them in the form of an array styled listing.

Another option is the creation of stacks. The stack library in Java allows the use of creating a stack of objects. The stack works on the Last In, First Out (LIFO) principle, when returning objects. This means that the latest item present on the stack will be the first one to be returned, therefore, earning the name of a stack, where you can take out the item on the top first.

There are five operations present that ensure that any vector can be created as a stack. They are as follows:

1. The first method `push()` is used to place any new object in the stack.
2. The second method `pop()` is used to remove an object from the stack.
3. The third method is to look at the top item of the stack.
4. The fourth method finds whether a stack is empty or not.
5. The final method finds a specific item in the stack and describes its positional distance from the top of the stack.

Remember, all stacks are empty when they are created in a Java implementation.

There is also an interface that describes a queue data structure. This is a collection where the items are ordered, but the order remains the one in which they are stored in the stack. All additions are added to the end of the object list, while all removals happen from the front. This is termed as *linked list implementation*. This presents the First In, First Out (FIFO) scheme, which provides a different implementation method, separating it from the LIFO option of the first type of stack.

So, to summarize what we have discussed so far, the following are a few basic operations needed to perform with collections:

1. Adding new objects to the collection.
2. Removing objects from the collection.
3. Looking for objects or group of objects in the collection.
4. Getting an object from the collection without removing it.
5. Looking for a specific object at a specific index.
6. Iterating through the collection one object at a time.

13.3.2 Benefits of Java Collections

The Java Collections Framework offers excellent benefits which are as follows:

1. The main advantage it offers is that it reduces the total programming effort that a developer must go through when creating long and complex programs. This is possible by setting up customized data structures and algorithms like custom collections that can be used as a plumbing system to deliver resources throughout the program.
2. It is important to create a strong data object system in programs. Whether you want to perform integration between various program elements or build operational capability between the different Java APIs, Java Collections provide you the opportunity to create adapter objects. In fact, it is also possible to come up with a conversion code, which allows a Java program to use different APIs and employ them to the optimal advantage of creating efficient and cost-effective programs.
3. The execution speed and the quality of the programming code are important parameters, especially for modern cloud applications. It is possible to increase the program efficiency by using the Collections Framework.

4. When you set up useful data structures in your program, you can come up with multiple interfaces. They are interchangeable because of the use of specific data collections. You have set up data structures that allow you to completely devote yourself to creating better code, while not worrying about how to tackle your objects and data storage elements. The collections allow you to create efficient data structures that can deal with all the information. They are also more flexible than simple arrays that are available in other various programming languages.
5. A key benefit is that unrelated APIs and programming elements can easily work together by using the strong collection interfaces. APIs can interact with each other and share information in the form of the collections that are empowered through the Java Collections Framework. You can employ node names and come up with a strong implementation solution, which is beneficial in a variety of environments.
6. Another advantage is that the use of collections available in Java make it easier to use new APIs and other tools. Naturally, collections can be used to pass data to the APIs as well as obtain the required inputs. This allows programmers to only have the knowledge of collections and then use any API that they require in a particular program. You do not have to learn from scratch every time there is a need to use a new API for a specific Java functionality. In fact, expert Java programmers create their own APIs to help other developers with code simplification and provide support for functions that may be required repetitively in a number of situations. There is no need to reinvent the wheel in terms of sharing objects, data, and information when providing operational capabilities in new APIs. All you need to do is make them compatible with the Java Collections Framework for the required level of connectivity and support.
7. The new data structures that are prepared in your programs can be made to conform to the collection interfaces in Java. This creates a database of objects and information that you can employ later as well. New algorithms can be easily made available for reuse in their implementation if they are prepared according to the needs of the collection interfaces compatible in Java.

13.3.3 Collection Interfaces

Java Collections interfaces form the foundation of the framework. They are often employed with the use of generic coding, which is then combined with a specific data collection. The interfaces that are created are perfect for reducing the runtime errors as all objects are checked during compilation. Java does not set up separate interfaces to ensure that the core collection is easily managed. It throws an exception if an unsupported operation is implemented using the Collections.

The interface here sets up the root of the collection hierarchy. There is a group of objects that make up the collection. The interface allows developers to employ various methods which allow them to check the elements in the collection, as well as perform the functions of removing, adding, and performing an iteration on the available collection elements.

Table 13.3 gives the list of core collection interfaces.

Table 13.3 List of core collection interfaces

Collection	Set	SortedSet
List	Map	SortedMap
Queues	ConcurrentMap	NavigableMap
ConcurrentNavigableMap		

Collections Framework has many collections but not all of them implement the Collection interface. For example, Map interface does not extend Collection interface or none of the classes implement the Collection interface.

The word Collection is little confusing in Java as there are three overloaded uses of the word "collection". Therefore, you should pay close attention to the word and make sure you refer it properly.

1. **Collection (note lowercase c):** This represents any of the data structures in which objects are stored and iterated over.
2. **Collection (note capital C):** This is the java.util.Collection interface. This interface is the one which is extended by List, Set, Queue, etc. Also, note that there is no direct implantation of Collection interface. Only other interfaces extend this interface.
3. **Collections (note "s" at the end and capital C):** This is not an interface but a class from the util package.java.util. Collections class. It contains static utility methods to use with collections. For example, sort, `emptyList`, `emptySet`, `emptyMap`, `addAll`, `binarySearch`, etc.

In this chapter, we will mainly talk about Collection interface. The following are different aspects of Collection interface:

1. **List:** This Collection interface represents list of things.
2. **Set:** This Collection interface represents unique things.
3. **Map:** This Collection interface represents things with a unique ID.
4. **Queue:** This Collection interface represents things arranged by the order in which they are to be processed.

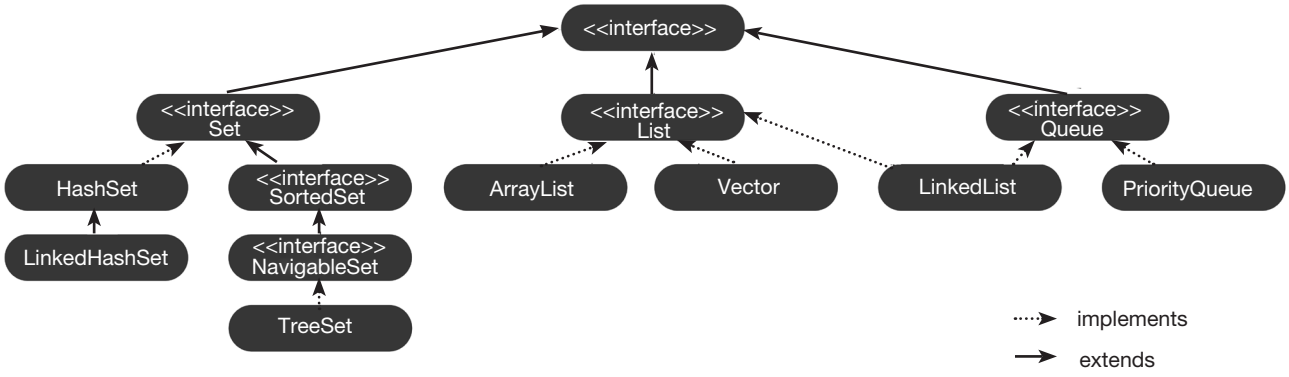


Figure 13.2 Collection Interface.

Figures 13.2 and 13.3 give us a good idea about some of the core interfaces and implantation classes. In Section 13.4, we will explore these classes in detail. The following are the subsets of these interfaces.

1. Sorted
2. Unsorted
3. Ordered
4. Unordered

The implemented classes can be one the following:

1. Unsorted and Unordered
2. Ordered but Unsorted
3. Ordered and Sorted

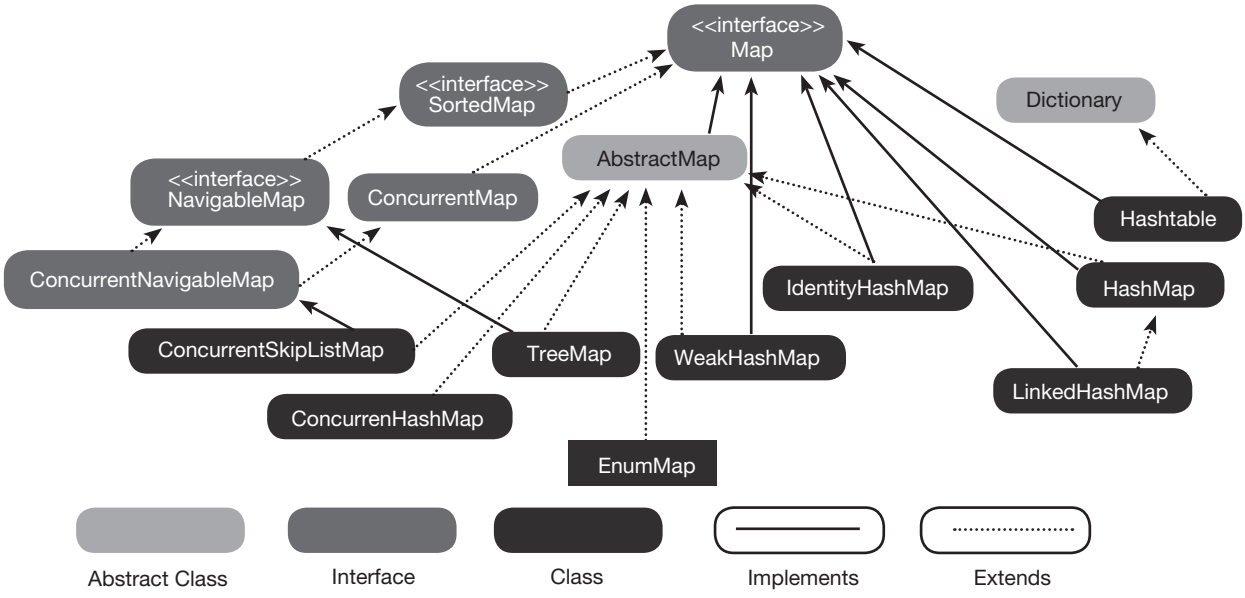


Figure 13.3 Map Interface.

It is important to note that an implemented class cannot be sorted but unordered. As you may have figured, this is because sorting itself is a kind of ordering. Now, let us take the example of HashSet, which is an unordered and unsorted set. On the other hand, the LinkedHashSet is ordered but not sorted. The order of LinkedHashSet is maintained on an insertion basis. The order is set as you add elements to it.

In order for us to understand the concept of collections, first we need to understand the concept of iteration. For a simple "for loop", we access elements in order of index such as 0, 1, 2, 3, etc. However, iterating over a collection is quite different. Elements are get collected one by one; so there is an order as it starts from the first element. However, in some cases, the first position is not defined the way we think. Take the example of Hashtable, in which there is no first, second, or third order. The elements are simply placed in a chaotic order based on the hashcode of the key. Hence, the first element is determined at the time of iteration because it constantly changes as the collection changes.

Let us explore Ordered and Sorted concepts.

1. **Ordered collection:** The collection is said to be ordered when iteration can happen in a specific order. As we have seen, Hashtable is not ordered since we cannot determine the order of the elements. The order is determined by the hashcode of the elements and constantly changes as the collection changes. On the other hand, ArrayList is ordered as it inserts the elements on specific indexes. You can determine which index to use to add an element to ArrayList. Hence, it is ordered via index position. However, LinkedHashSet keeps the insertion order; that means that the first element inserted will be on the first order and last element inserted will be on the last order. Some collections use the natural order; that means that they are not only ordered but sorted (by natural order, such as A, B, C, … and 1, 2, 3, …). Now, natural order is very straightforward in case of alphabets and numbers, but what about objects? Objects do not have a natural order. So how do we determine the objects' natural order? For this, Java provides an interface called *Comparable*. This interface has a provision to define a natural order. It means, developers have the freedom to define the natural order for the objects. This rule can be like deciding order based on one of the instance variables, for example, price, age, name, etc. Java also provides another interface called *Comparator*, which can be used to define sort order for objects.
2. **Sorted collection:** This type of collection is ordered based on some rule(s). These rules are called *sort order*. This sort order is not determined by the time the object gets inserted into the collection, or accessed the last time, or on which position it was inserted. This sorting is based on the properties of the objects that are added to the collection. It means that we simply insert the object into the collection and let the collection determine the order of that element based on the sort order. A collection that keeps an order like ArrayList are not really sorted unless they are explicitly sorted based on some type of sort order. In most cases, the sort order is the natural order of that object.

13.3.4 Collections Classes

Java provides amazing classes that you can employ to carry out the required functions in your program. The main class is the HashSet class. It is backed by the API of HashMap. It does not provide an iteration order and is capable of implementing bucket lists. The initial capacity can be set up in this class, ensuring that it follows a fixed set of capacity increments that save the available program resources. It provides a constant time performance where the elements are dispersed properly in the available buckets.

The TreeSet Class makes use of a TreeMap. The natural order to the creation time is often employed when constructing this Collections class. This class is employed when we want to maintain a consistent structure. It employs comparisons to check different elements, ensuring that it offers the use that we are looking for in a class. Table 13.4 lists some of the top options that are available in Java for use in Collections.

Table 13.4 List of Collections Classes

Map	Set	List	Queue	Utilities
HashMap	HashSet	ArrayList	PriorityQueue	Collections
Hashtable	LinkedHashSet	Vector		Arrays
TreeMap	TreeSet	LinkedList		
LinkedHashMap				

Table 13.5 shows all the classes and indicates whether they are ordered and/or sorted.

Table 13.5 Properties of Collection Classes

Class	Implements	Ordered	Sorted
HashMap	Map	No	No
HashTable	Map	No	No
TreeMap	Map	Sorted	Sorted by natural order. It can also be sorted by custom comparison rules.
LinkedHashMap	Map	Ordered by insertion order. It can also be ordered by last access order.	No
HashSet	Set	No	No
TreeSet	Set	Sorted	Sorted by natural order. It can also be sorted by custom comparison rules.
LinkedHashSet	Set	Ordered by insertion order	No
ArrayList	List	Ordered by index	No
Vector	List	Ordered by index	No
LinkedList	List	Ordered by index	No
PriorityQueue	Queue	Sorted	Sorted by to-do order

13.4 | Implementing Collection Classes

There are several ways to implement Java Collections functionality. Here, we will explain each concept to build up the desired level of knowledge required for using Collections with Generics. We will describe various API settings that allow the use of collection functionality and provide a detailed discussion on how these concepts are implemented in the language.

13.4.1 Map Interface

Map is a key value pair collection. It takes unique identifier as a key that acts like an ID. Both key and value take object as input. This allows us to search for a value based on the key. Maps and Sets both rely on the `equals()` method to check if the two keys are the same or different.

Figure 13.4 shows how values are stored as key–value pair in Map. Hashcode buckets contain the keys and point to its corresponding values. In the following example, key 315 is associated with value "Parle Biscuit", 612 is associated with "Parlet Chocolate", etc.

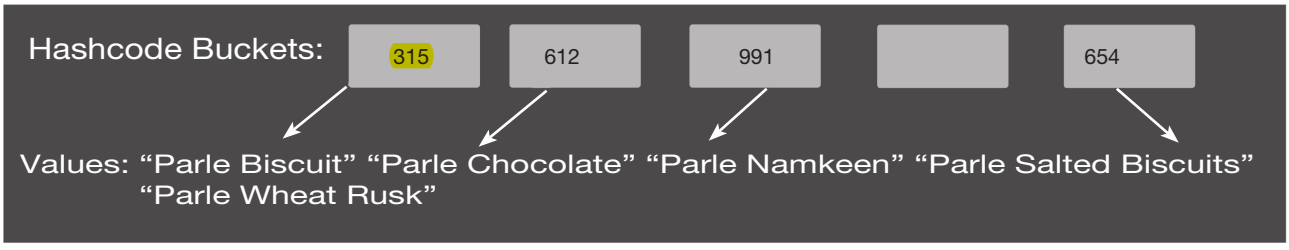


Figure 13.4 Key–value structure in Map.

13.4.1.1 HashMap

HashMap is a collection class that uses the system of pairs, where one is the key and the other is the corresponding value. The syntax is in the form of `HashMap<K,V>`, where the key comes first while it is followed up by the corresponding value. The objects that are stored in this collection do not have to be ordered as it is employed to find any value by using the corresponding key. It is possible to set up null values in this collection class.

This class is imported from java.util.HashMap, which describes its path in the utility library. The following is an example of using this class, in which we have removed the importing of the various other requirements for the program:

```
package java11.fundamentals.chapter13;

import java.util.HashMap;
import java.util.Iterator;
import java.util.Map;
import java.util.Set;

public class HashMapExample {
   public static void main(String[] args) {
       HashMap<Integer, String> hmap1 = new HashMap<Integer, String>();
       hmap1.put(14, "George");
       hmap1.put(33, "Paul");
       hmap1.put(16, "Jane");
       hmap1.put(7, "Brian");
       hmap1.put(19, "Jack");
       Set set1 = hmap1.entrySet();
       Iterator iterator1 = set1.iterator();
       while (iterator1.hasNext()) {
           Map.Entry ment1 = (Map.Entry) iterator1.next();
           System.out.println("The value is: " + ment1.getValue() + " and key is: " +
ment1.getKey());
       }
       String va = hmap1.get(2);
       System.out.println("Index 2 has value of " + va);
       hmap1.remove(16);
       Set set2 = hmap1.entrySet();
       Iterator iterator2 = set2.iterator();
       while (iterator2.hasNext()) {
           Map.Entry ment2 = (Map.Entry) iterator2.next();
           System.out.println("Now value is " + ment2.getValue() + "and key is: " +
ment2.getKey());
       }
   }
}
```

The above program produces the following output.

```
The value is: Jane and key is: 16
The value is: Paul and key is: 33
The value is: Jack and key is: 19
The value is: Brian and key is: 7
The value is: George and key is: 14
Index 2 has value of null
Now value is Pauland key is: 33
Now value is Jackand key is: 19
Now value is Brianand key is: 7
Now value is Georgeand key is: 14
```

This is a detailed program that shows the complete use of the HashMap class. All properties are perfectly shown by the use of the example. The first section sets up a HashMap collection and places five values with five keys in it. They are displayed using the while conditional loop and follow an alphabetical pattern according to the values, which is possible with the use of the iterator. Next, we remove the value associated with the key of 16 which will remove the "Jane" entry. Now, when we perform

the same operation, we will find the list printed only four values. This time the "Jane" value and its key is not present in the collection.

Can you insert integer as a key and image as value in HashMap?

There are some other excellent methods. The `clear()` method can clear a Map collection completely, while the `clone()` method creates a clone of one map to another, which can then be manipulated while ensuring that we also keep a legacy copy of the original collection. These represent some basic use of this class, allowing us to experiment with this type of collection, where the purpose is to create an unordered pair of object values.

13.4.1.2 Hashtable

Another collection class which is commonly implemented is that of the Hashtable. It is assigned from the java.util.Hashtable importing tree. It offers a slightly different implementation than the map, as it creates a table of keys and values, resulting in the production of synchronized set of objects, just like a truth table or a cartesian coordinate system.

It is possible to initiate it using the default constructor or to set up its size. It is also possible to read the elements of a Map in a Hashtable. Considering the earlier example in the HashMap given in Section 13.4.1.1, here is a program that describes the use of a Hashtable in Java:

```
package java11.fundamentals.chapter13;

import java.util.Enumeration;
import java.util.Hashtable;

public class HashtableExample {
    public static void main(String args[]) {
        Enumeration nms;
        String keys;
        Hashtable<String, String> hashtable = new Hashtable<String, String>();
        hashtable.put("Key1", "Adam");
        hashtable.put("Key2", "Brian");
        hashtable.put("Key3", "Charles");
        hashtable.put("Key4", "Dean");
        hashtable.put("Key5", "Peter");
        nms = hashtable.keys();
        while (nms.hasMoreElements()) {
            keys = (String) nms.nextElement();
            System.out.println("Key is " + keys + " & value is " + hashtable.get(keys));
        }
    }
}
```

The above program produces the following output.

```
Key is Key4 & value is Dean
Key is Key3 & value is Charles
Key is Key2 & value is Brian
Key is Key1 & value is Adam
Key is Key5 & value is Peter
```

In the above output, you can see the keys are printed in descending order, from Key4 to Key1, with the corresponding values. The last key entered is always presented at the end. There are various methods that are available for use in this collection class. It is possible to create an enumeration of the keys present in the table, while the `rehash()` method can enhance the size of the table and rehash all its keys. Both keys and the values can be searched within the collection, and it is also possible to get the return of the elements that are present as values in the table.

This class is excellent for use in a variety of conditions. It is perfect when we need to perform collection comparisons to find out something has changed. The table objects can be directly printed by giving the name of the hashtable identifier directly in the print statement.

13.4.1.3 TreeMap

TreeMap is a class which implements a navigable map in Java. It employs the natural ordering of the keys that are used to enter values. It is different from the HashMap since it creates a map according to the ascending order of its key values. However, this means that this class is not thread-safe and therefore, cannot be properly used with parallel programming in Java, unless it is kept safe.

Here, we present an example to show this behavior as compared to the one that we presented in HashMap. This will allow you to understand the use of this collection class, which is perfect in several scenarios where the specific order is important.

```
package java11.fundamentals.chapter13;

import java.util.Set;
import java.util.Iterator;
import java.util.Map;
import java.util.TreeMap;

public class TreeMapExample {
    public static void main(String[] args) {
        TreeMap<Integer, String> trmap = new TreeMap<Integer, String>();
        trmap.put(1, "Object 1");
        trmap.put(17, "Object 2");
        trmap.put(50, "Object 3");
        trmap.put(7, "Object 4");
        trmap.put(3, "Object 5");
        Set set = trmap.entrySet();
        Iterator iterator1 = set.iterator();
        while (iterator1.hasNext()) {
            Map.Entry ment = (Map.Entry) iterator1.next();
            System.out.print("key is: " + ment.getKey() + " and Value is: ");
            System.out.println(ment.getValue());
        }
    }
}
```

The above program produces the following output.

```
key is: 1 and Value is: Object 1
key is: 3 and Value is: Object 5
key is: 7 and Value is: Object 4
key is: 17 and Value is: Object 2
key is: 50 and Value is: Object 3
```

This program principles out objects not based on their value and setting. They are printed out with the help of the order of the key values. This means that the TreeMap which is created for the collection automatically comes up with the key order of 1, 3, 7, 17, and 50, which will correspond to their specific values and will be printed in that order.

13.4.1.4 LinkedHashMap

The next implementation class is the LinkedHashMap. It can be presented as a combination of the facilities that are present in a hash table and then combined with the map interface that allows for creating a predictable iteration order for use in Java programs. It is different from a normal HashMap, because it uses a double linked list which connects to the entries that are present in this collections class. The linking defines the way iteration will be performed on the keys that are inserted into the map.

The added functionality provides a situation where we need access the insertion order of the values that become part of the collection. Here is an example to help you understand the use of this Java Collections class:

```
package java11.fundamentals.chapter13;

import java.util.LinkedHashMap;
import java.util.Iterator;
import java.util.Map;
import java.util.Set;

public class LinkedHashMapExample {
    public static void main(String[] args) {
        // Declaring a HashMap
        LinkedHashMap<Integer, String> lihamap = new LinkedHashMap<Integer, String>();
        // Adding the elements to this collection map
        lihamap.put(21, "Abe");
        lihamap.put(35, "Drown");
        lihamap.put(1, "Jack");
        lihamap.put(3, "Karen");
        lihamap.put(100, "Lin");

        // Generating the required set
        Set set1 = lihamap.entrySet();

        // Displaying elements from this collection map
        Iterator iter1 = set1.iterator();
        while (iter1.hasNext()) {
            Map.Entry me = (Map.Entry) iter1.next();
            System.out.print("The key is: " + me.getKey() + " and Value is: " +
me.getValue() + "\n");
        }
    }
}
```

The above program produces the following output.

```
The key is: 21 and Value is: Abe
The key is: 35 and Value is: Drown
The key is: 1 and Value is: Jack
The key is: 3 and Value is: Karen
The key is: 100 and Value is: Lin
```

The output of this program is different from previous examples. It will return and print the entire list in the order of the way the values are inserted in the program. The first value printed will be Abe with the last value printed will be Lin, in the same order. It is possible now to create chronological lists that you can use for the required functionality in your program. Remember, this map interface requires unique keys and it maintains the insertion order.

There are obvious benefits of using LinkedHashMap. Think of a situation where one module reads a map and copies it, and then produces results that completely depend on the order of the map elements it received.

This map provides access to all the basic operations on collection objects. It may have slower performance than a simple HashMap due to managing the order of the collections. However, performing an iteration is faster since it is only based on the size of the map, as we need access only to the order of the total entries.

Another important element to note is that the LinkedHashMap is not synchronization safe, since a thread can modify the map and affect the order of the stored collections. The knowledge of this type of map is important since it allows programming using Java to make informed decisions and pick the best options from the available set of services.

13.4.2 Set Interface

Set does not allow duplicates. Hence, each element is unique in a set. Similar to Map, Set also uses `equals()` method to check if two objects are equal. In case two identical objects found, only one of them will get inserted into the Set. Let us explore the concept of Set implementation. Figure 13.5 shows the representation of Set.

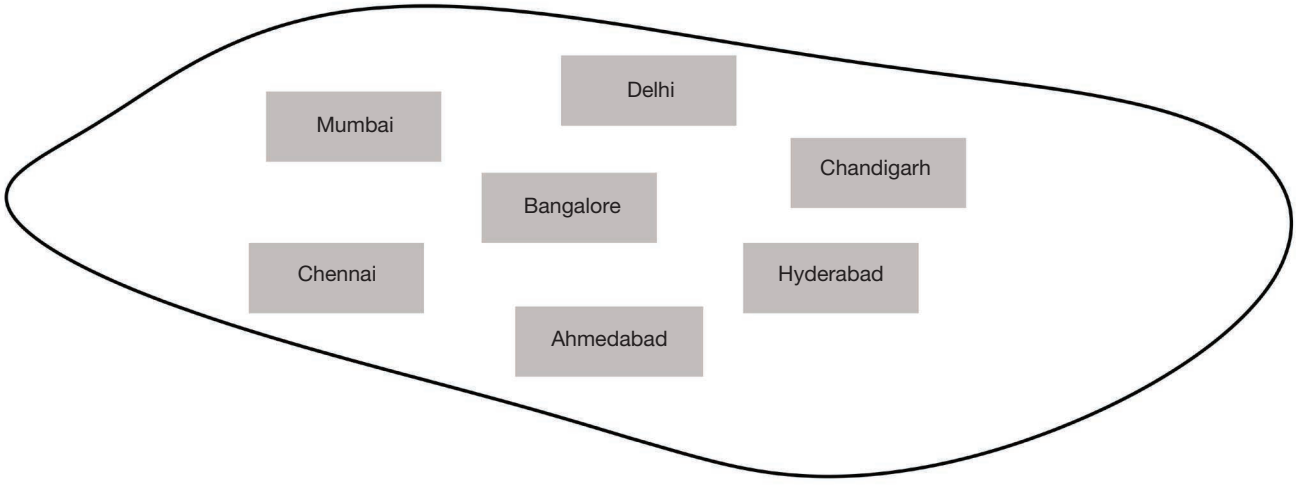


Figure 13.5 Representation of Set.

13.4.2.1 HashSet

Now that we have gone through the classes that employ the map interface, we will discuss the set-based classes that can be implemented from the Set interface in Java. This class does not follow any order; this means that the elements are returned in a random order. It also does not allow the use of duplicate values, as inputting the same value will simply replace the older one.

This set collection also accepts null values, but there can only be one null value stored in a HashSet. The iterator is fail-safe and therefore, it is not possible to modify the set once the iterator has worked on it. However, the remove method within the iterator can remove any value from the hash set without a problem.

The best way to understand the use of a HashSet is to go through a simple example. Here, we describe how programmers can use it as well as display the different items stored in the collection, which are easier to do when compared with the maps that require handling of the keys as well.

```
package java11.fundamentals.chapter13;

import java.util.HashSet;

public class HashSetExample {

    public static void main(String[] args) {

        // Declaring a HashSet
        HashSet<String> haset = new HashSet<String>();

        // Adding different elements including null ones
        haset.add("Apricot");
        haset.add("Mango");
        haset.add("Orange");
        haset.add("Strawberry");
        haset.add("Dates");

        // Adding duplicate elements
        haset.add("Orange");
        haset.add("Mango");

        // Multiple null values
        haset.add(null);
        haset.add(null);

        // Displaying the stored HashSet elements
        System.out.println(haset);
    }
}
```

The above program produces the following output.

```
[null, Strawberry, Mango, Apricot, Dates, Orange]
```

The printing of the set will reveal that all values will be displayed without any order. There will only be a single iteration of both the duplicates as well as the null values that we add twice. They are treated similarly, only producing a single record in the HashSet. This is a simple test, which for some can look like the use of an array. However, HashSet implementation offers other advantages by providing methods that allow programmers to manipulate the stored object values.

It is also a good method in terms of the resources that it requires during the operations. It has good performance and works well with the handling of the load that it often possesses. The memory overhead is often not much, and the searching and rehashing operations can use more resources during its application.

Can you get a sorted data from HashSet?

13.4.2.2 LinkedHashSet

Since you have understood the use of a basic HashSet, it is time to discuss the LinkedHashSet class in Java. It also contains unique elements but is different since it maintains the insertion order of the elements added in the class. The sorting of elements are important, especially when a LinkedHashSet must operate with other programming elements, where a change in order can produce a different output altogether.

The double linked list is created for all the elements. It is designed for circumstances where you need to employ an iteration order on the added object values. Using the iterator ensures that the same order is returned in which the elements are added to this hashed set, which has linked elements. Here is an example that shows the use of these practices.

```
package java11.fundamentals.chapter13;

import java.util.LinkedHashSet;

public class LinkedHashSetExample {

    public static void main(String[] args) {

        // Creating a LinkedHashSet for String
        LinkedHashSet<String> lhset1 = new LinkedHashSet<String>();

        // Adding different elements to the LinkedHashSet
        lhset1.add("Z");
        lhset1.add("R");
        lhset1.add("M");
        lhset1.add("O");
        lhset1.add("KKK");
        lhset1.add("EFG");

        System.out.println(lhset1);

        // Now creating a LinkedHashSet for Integer
        LinkedHashSet<Integer> lhset2 = new LinkedHashSet<Integer>();

        // Adding integer elements
        lhset2.add(95);
        lhset2.add(13);
        lhset2.add(0);
        lhset2.add(55);
        lhset2.add(33);
        lhset2.add(61);

        System.out.println(lhset2);
    }
}
```

The above program produces the following output.

```
[Z, R, M, O, KKK, EFG]
[95, 13, 0, 55, 33, 61]
```

Printing will start from Z and end with EFG as it was the last item added to the LinkedHashSet. The second LinkedHashSet follows the same method, where it starts from 95 and then continues printing to the last added value of 61. The double linking ensures that it is not possible to insert a duplicate element in this iteration scheme.

It is also possible to remove items. A removal simply updates the iteration, as another system print will find that the item is removed and the item next to it is updated in the order of the storage of the objects present in the set collection. Creating sets are different from the maps as there are no keys that can be used to direct towards specific values.

Linked collection classes use additional CPU resources and require more storage. The simple classes of HashSet and HashMap work well if there is no need to remember the order of the values that are inserted in the program. However, the LinkedHashSet is perfect when you want to maintain the organization of the order of the values, and then set up various objects that may allow you to store, share, and print information sets in your Java programs.

13.4.2.3 TreeSet

As already described with TreeMap, the TreeSet is a class that stores the order of the elements in an ascending order. It also allows you to include null elements, while it is also not synchronized. Although synchronization is possible by explicitly setting it up as a sorted set from the Java Collections Framework. Here is a simple example to show how it works before we discuss some of this class's implementation advantages in specific situations.

```
package java11.fundamentals.chapter13;
import java.util.TreeSet;
public class TreeSetExample {
    public static void main(String args[]) {

        // Creating a String type TreeSet
        TreeSet<String> tset = new TreeSet<String>();

        // Adding various string elements to the above TreeSet
        tset.add("EFG");
        tset.add("Stores");
        tset.add("Tests");
        tset.add("Pens");
        tset.add("Ink");
        tset.add("Jane");

        // Displaying the collection of TreeSet elements
        System.out.println(tset);
    }
}
```

This program will print the strings in an ascending order. The print line will display the following.

```
[EFG, Ink, Jane, Pens, Stores, Tests]
```

This shows that printing produces the display in ascending order. This function not only works well with String objects, it is also perfect for use in a variety of Integer objects environments where the order of the collection is of paramount importance. As you can see, it does not allow for repeating values as this will make it impossible to use a sorting scheme.

It is possible to specify the sorting order that you would like to use with this collection class. Also, it cannot hold objects that belong to different primitive types. This means that if a TreeSet has string values in it, it will throw a class exception when a code attempts to add an integer to the collection and vice versa.

Since this tree set can use a comparator to set up the order of the elements, it can be useful in a variety of conditions where the data is randomly collected. However, its use requires that the elements are sorted and are available for use according to a specific order, which is possible by using a comparator when defining the collections class.

13.4.3 List Interface

List is all about index. It offers various methods related to index, such as get(int index), indexOf(Object o), and add(int index, Object obj). List implementations are ordered by index. An object added to List will get added to the end of the list unless you specify a specific index position. Figure 13.6 shows the implementations of List.

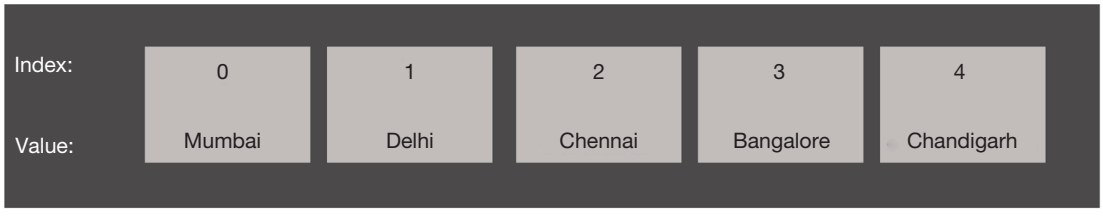


Figure 13.6 Implementation of List.

13.4.3.1 ArrayList

The ArrayList is also a part of Java Collections. It is designed to allow programmers to use dynamic array collections. It is obviously slower than using simple arrays, but it is perfect for use when you have to perform multiple array manipulations. It belongs to the AbstractList class and therefore employs the list interface. Figure 13.7 shows the representation of ArrayList.

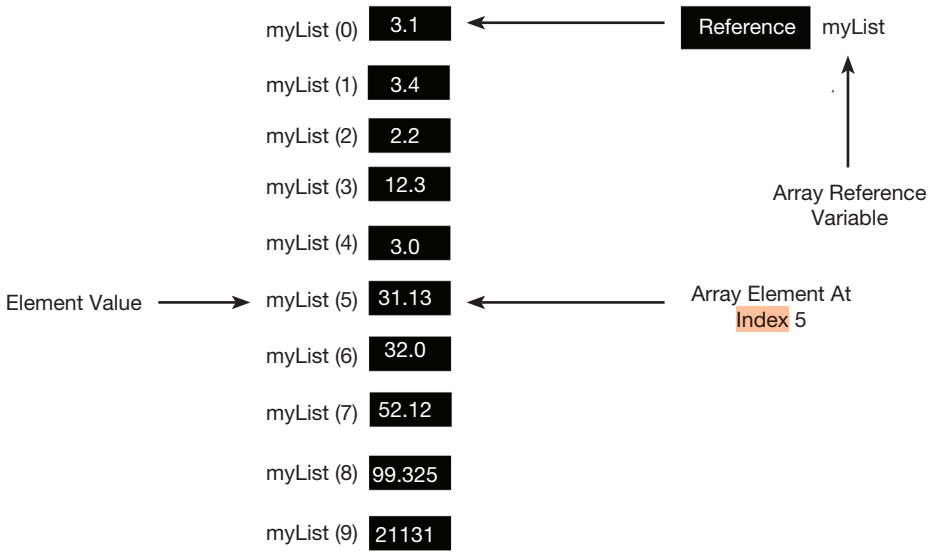


Figure 13.7 Representation of ArrayList.

This class is designed to provide random access to the list of array elements. It cannot be used for primitive types. It is designed for use in the same manner as vectors are used in C++ programming language. It can contain duplicate elements as well, while it also maintains the insertion order of inputted values.

Similar to other collection classes, ArrayList is a non-synchronized class. Random access is made possible because the array operates on the basis of index, which eliminates the need to iterate to reach each item. All object manipulations are slow because there is a strong shifting which occurs when a single element is removed from the ArrayList. This list is easy to create and the available methods allow programmers to add and remove elements, while also ensuring that it is possible to use the index values of the stored array elements.

Here is a program that shows how this class can be implemented in your Java program:

```
package java11.fundamentals.chapter13;
import java.util.*;
import java.io.*;
public class ArrayListExample {
    public static void main(String[] args) {
        // First setting the size of ArrayList
        int size = 5;
        // Now declaring ArrayList with that size
        ArrayList<Integer> arrlist = new ArrayList<Integer>(size);
        // Now Appending new elements in the list
        for (int i = 1; i <= size; i++) {
            arrlist.add(i);
        }
        System.out.println(arrlist);
        arrlist.remove(2);
        // Again displaying the ArrayList after removing
        System.out.println(arrlist);
    }
}
```

This is a simple program, which will first store the first five integers in the ArrayList by the name of "arrlist". The first print will include [1, 2, 3, 4, 5]. These values are stored sequentially just like a normal one-dimensional array indexing. This means that removing the index 2 will remove the third value from the list, which is 3 in this example. The next print will show the following output.

```
[1, 2, 3, 4, 5]
[1, 2, 4, 5]
```

This shows that it is easy to use, manipulate, and set up for different functions. Setting up array lists with fixed initial capacity is the ideal way to go about it. It ensures that you have greater control over your program and can reliably use your lists to create inputs for other program elements.

Is ArrayList safe to use in a multithreaded environment?

13.4.3.2 Vector

The Vector class in Java belongs to the List hierarchy, and works like a growing array of objects. All the stored objects are accessible using an integer index. It is named as a vector as the size of this class varies according to the items that are currently present in the Vector. It provides storage management by setting up a fixed capacity as well as allowing a capacity increment to ensure that it is always possible to store new objects in it.

The advantage of using this class is that it is synchronized and therefore, can be used in all types of Java programs without ever worrying about parallel processing problems. In fact, the dynamic array created in this class often contains legacy methods, which go beyond the scope of the Java Collections Framework.

Vector is great for use, if you are programming for predictable situations where you do not know the size of the required arrays. It is great for situations where you may want arrays that can smartly change their size and always use only the minimal program sources. There are various methods present in this class, which allow programmers to carry out the intended programming. Here is an example:

```
package java11.fundamentals.chapter13;
import java.util.*;
public class VectorExample {
    public static void main(String[] args) {
        // Setting up initial size and increments
        Vector vec = new Vector(3, 2);
        System.out.println("Initial size is: " + vec.size());
        System.out.println("Initial capacity is: " + vec.capacity());
        // Adding elements
        vec.addElement(new Integer(1));
        vec.addElement(new Integer(2));
        vec.addElement(new Integer(3));
        vec.addElement(new Integer(4));
        System.out.println("The capacity after four additions is: " + vec.capacity());
        vec.addElement(new Double(6.55));
        System.out.println("Now capacity is: " + vec.capacity());
        vec.addElement(new Double(5.35));
        vec.addElement(new Integer(8));
        System.out.println("Now capacity is: " + vec.capacity());
        vec.addElement(new Float(9.5));
        vec.addElement(new Integer(10));
        System.out.println("Now capacity is: " + vec.capacity());
        vec.addElement(new Integer(11));
        vec.addElement(new Integer(12));
        System.out.println("First element is: " + (Integer) vec.firstElement());
        System.out.println("Last element is: " + (Integer) vec.lastElement());
        if (vec.contains(new Integer(3))) {
            System.out.println("Vector contains 3.");
        }
        // enumerate the vector elements
        Enumeration vecEnum = vec.elements();
        System.out.println("\nElements in the vector:");
        while (vecEnum.hasMoreElements()){
            System.out.print(vecEnum.nextElement() + " ");
        }
        System.out.println();
    }
}
```

The above program will produce the following output.

```
Initial size is: 0
Initial capacity is: 3
The capacity after four additions is: 5
Now capacity is: 5
Now capacity is: 7
Now capacity is: 9
First element is: 1
Last element is: 12
Vector contains 3.

Elements in the vector:
1 2 3 4 6.55 5.35 8 9.5 10 11 12
```

This is a detailed program that uses several ways in which the vector is created and employed. We first set up a vector with a capacity of 3 and an increment of 2. However, when we initially find the size of the vector, it is zero due to the absence of any stored value but still showing that it can accept three elements.

We add four elements in the Vector, which will put it above its initial capacity. This invokes an increment of 2, which is shown when the next output of the Vector capacity will show 5 as the number. The capacity is printed two more times, where it will show 7 and 9 with 2-place increments as set up.

We then show that it is possible to find the first and last element that will be stored in the same order in which we have stored them, as there is no hash function performed on them. It is also possible to check for specific elements within the vector using a test performed with the `contains()` method. An enumeration can be employed to read the values from the vector. This can then be displayed on the console.

Vector has its special application when we want to use multiple threads. However, since it is synchronized it can be slow in terms of adding and removing elements, as well performing a vector element search. It is good practice to mention the increment that you want, otherwise the default increment is to double the initial size, which can be wasteful in terms of resources.

13.4.3.3 LinkedList

Java provides another class from the List interface of LinkedList. This class provides double linking to store the elements. It can contain duplicate elements, while having the capacity to maintain the insertion order. As we have observed with similar classes, it is not designed for synchronized use. It is quick to manipulate as no object shifting is required with the double linked listing structure. Figure 13.8 shows representation of LinkedList.

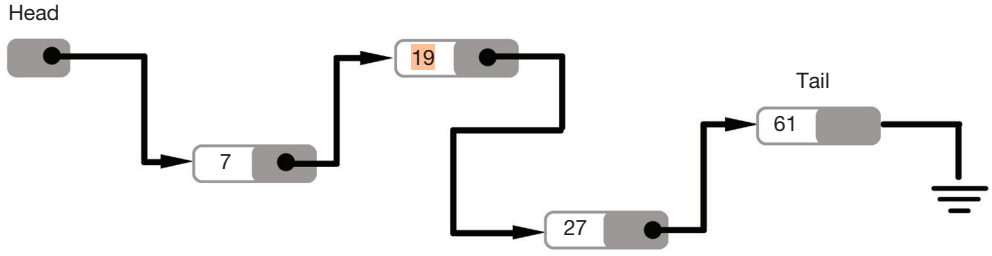


Figure 13.8 Representation of LinkedList.

LinkedList also implements the Deque interface and therefore, it can be used in a variety of ways. It is possible to use this class for stacking or creating a queue of objects. With double links, it is possible to remove the items from the start and end of the linked list. Although it is possible to add or remove items in numerous ways, access to the elements is only available in a sequential manner, where the search can occur either in a forward or reverse direction and will take time according to the position of the specific stored element.

LinkedList is excellent for manipulating the order of elements. This allows setting up elements for other program sections. Here is an example that shows the details of this use:

```
package java11.fundamentals.chapter13;
import java.util.LinkedList;
public class LinkedListExample {
    public static void main(String[] args) {
        LinkedList<String> obj = new LinkedList<String>();
        // Adding elements in various orders and positions
        obj.add("a");
        obj.add("b");
        obj.addLast("c");
        obj.addFirst("d");
        obj.add(2, "e");
        obj.add("f");
        obj.add("g");
        System.out.println("Linked list is: " + obj);
        // Now removing elements from the linked list using different options
        obj.remove("b");
        obj.remove(3);
        obj.removeFirst();
        obj.removeLast();
        System.out.println("New linked list after removing: " + obj);
        // Finding elements in the linked list
        boolean stat = obj.contains("e");
        if (stat) {
            System.out.println("List contains the element 'e' ");
        }else {
            System.out.println("List doesn't contain the element 'e'");
        }
        // Other linked list information
        int size = obj.size();
        System.out.println("Size of linked list = " + size);
    }
}
```

The above program produces the following output.

```
Linked list is: [d, a, e, b, c, f, g]
New linked list after removing: [a, e, f]
List contains the element 'e'
Size of linked list = 3
```

This is an excellent example as it shows that we can add to LinkedList in a variety of ways. If we add elements normally, they follow their insertion order and are indexed starting from 0. However, as already discussed, it is possible to add at the top or the bottom of the list, which will affect the entire list (e.g., we did that when we added "c" and "d").

We also added "e" at a particular index, which will move the other items in the list by one position. Similarly, it is also possible to remove the items. We get much greater control when dealing with LinkedList. We can use the element value or its index position to look for the element. We can also remove the first or the last element in the LinkedList. The size of this list will be according to the items that we have currently stored in this collections class.

13.4.4 Queue Interface

Queue, as the name suggests, holds things that need to be processed in sequence, such as FIFO or LIFO. This structure can be useful for to-do types of list. Queues offers special methods to add, remove, and review elements in addition to all the standard Collection methods. Let us see the implementation class of Queue.

13.4.4.1 PriorityQueue

PriorityQueue is a class that belongs to the Queue implementation, which is known for following the FIFO model. However, there are times where we need to perform processes or prioritize the elements. This is possible by using the PriorityQueue Class, which allows developers to set up priority processing operations.

This class is perfect for creating programs that may have to serve based on the priority of the available information. An example would be in a financial situation, where a company wants its premium customers to receive their processed information first. The PriorityQueue class implementation in Java will work perfectly in such collection scenarios.

This class does not allow for null values, as they are not in line with the function of this collection. It also does not allow you to create objects that cannot be compared with each other since this is essential for setting up the priority. This class uses the natural order of the elements, or employs the comparator that you have set up for the queue.

The smallest element will always be the head (the first element) of this class according to the required ordering. If there is a tie for a specific position, the class will arbitrarily place the elements, so it is best to avoid ties altogether. The queue retrieval operations also work by accessing the element at the head of the queue. It has methods from several structure trees including ones from Object, Collection, AbstractCollection, and AbstractQueue. This class is not thread-safe, but a multithreading version of the class is available for use these days. Here is an example to show how this Java Collection class is used:

```
package java11.fundamentals.chapter13;
import java.util.*;
public class PriorityQueueExample {
    public static void main(String[] args) {
        // Creating the empty priority queue
        PriorityQueue<String> prQueue = new PriorityQueue<String>();
        // Now adding the items
        prQueue.add("C");
        prQueue.add("Java");
        prQueue.add("Python");
        prQueue.add("C++");
        // Printing the most priority element
        System.out.println("The head value by using peek function is: " + prQueue.peek());
        // Now printing all elements
        System.out.println("The total queue elements are:");
        Iterator itr1 = prQueue.iterator();
        while (itr1.hasNext()){
            System.out.println(itr1.next());
        }
        // Now removing the top priority element (or head of queue)
        // And printing the modified PriorityQueue
        prQueue.poll();
        System.out.println("After removing an element with poll function: ");
        Iterator<String> itr2 = prQueue.iterator();
        while (itr2.hasNext()) {
            System.out.println(itr2.next());
        }
        // Removing one value of Java
        prQueue.remove("Java");
        System.out.println("after removing Java with remove function:");
        Iterator<String> itr3 = prQueue.iterator();
        while (itr3.hasNext()){
            System.out.println(itr3.next());
        }
        // Checking a particular element in the PriorityQueue
        boolean a = prQueue.contains("C");
        System.out.println("Does this priority queue contains C: " + a);
    }
}
```

The above program produces the following output.

```
The head value by using peek function is: C
The total queue elements are:
C
C++
Python
Java
After removing an element with poll function:
C++
Java
Python
after removing Java with remove function:
C++
Python
Does this priority queue contains C: false
```

Here, we set up a priority queue which holds the names of the programming languages. Then, we remove an element from the head of the list. This means that now the available values are rearranged. We then remove a specific value as well, and then run a search to check for an item to be present in the collection.

There are various situations in which these priorities can be really helpful during their use in the program. They are perfect for implementing logical decisions as well; for example, when requiring a reading of objects from an available collection.

13.4.5 Stream API

Java has an updated Stream API, which has been significantly improved over its previous version. It is designed to help developers improve their set of aggregate operations, which require the use of a sequence of objects. Stream API is a sequence of operations, where each operation can be set up in the stream. It is designed to produce a stream of objects according to the parameters that are set up in a filter element.

Object references are passed on in a stream, which result in producing a total weight of the objects. Stream operations can belong to various primitive types such as integer, long, and double. They use a pipeline that contains a source, which can be a collection of the several classes that we have discussed above, or it may constitute other elements. It is a lazy implementation where computation is only performed as and when required.

The use of stream is quite similar to a collection but is different in terms of their objectives. Collections are designed to improve the operational control required for managing the elements within them. There are terminal operations that are available for use in this API. Stream operations are great in terms of allowing the program to describe its source material and then defining the required computer operations that must be performed on them.

The iterator method is also available for use. It is possible to run queries on stream sources, while they also allow programmers to perform concurrent modifications. Java streams are designed to provide a sequence of elements on which operations can be performed on demand, differentiating them from the collections that are available in the programming language.

Stream provides both operations of pipelining and internal iterations. Here is a simple example of how different objects and collections can be passed to a stream and then work with some operations:

```
Stream.of("a1", "a2", "a3")
.findFirst()
.ifPresent(System.out::println);
```

This stream records the array as a list, and then the stream allows it to be captured in terms of the operations that are required, such as bringing out the first stored item. There are several methods that are helpful when employed with the streams in Java. Take the example of forEach, which works like a loop in the programming language.

The forEach loop is great at running an iteration of the items that must belong to a particular category or fulfill a specific condition from the group mentioned within the statement. This loop uses an internal iterator and can employ the interface of a Consumer for improved functionality. Here is an example:

```
package java11.fundamentals.chapter13;

import java.util.ArrayList;
import java.util.List;
import java.util.function.Consumer;

public class StreamExample {

       public static void main(String args[]){
           List<String> names = new ArrayList<>();

           names.add("Harry");
           names.add("Steve");
           names.add("Adams");
           names.add("Chris");
           names.add("Allen");
           names.forEach(new Consumer<String>() {
               public void accept(String name) {
                   System.out.println(name);
               }
           }
       );
   }
}
```

The above program produces the following output.

```
Harry
Steve
Adams
Chris
Allen
```

This adds various names and then carries out a loop which accepts and prints out the names from the list. This is a convenient loop which we can employ in place of a "for" loop, while it can cater to the different ways in which we can employ collections in the program loops. There are other methods that are available in collections and streams.

The list classes have various useful methods that can be combined when employed with object streams. It is possible to use the `removeIf()` method, which is designed to remove an object only if a certain condition is met. Sorting is also possible. These are all present in the *java.util* library and it is ideal to include import java.util.*; to ensure that complete functionality will be possible in a Java snippet or a complete program.

13.5 | List of Key Methods for Arrays and Collections

The methods covered in following subsections are key methods you will be using frequently in your programs. You should remember these methods so that you can find solutions to the problems you may face while using Arrays and Collections.

13.5.1 Arrays (java.util.Arrays)

Table 13.6 lists methods that are useful for Arrays. These methods can be used to work on data using Arrays.

Table 13.6 List of Arrays Methods

Method	Description
static List asList(T[])	This method is helpful for converting and binding Array to a List.
static int binarySearch(Object [], key) static int binarySearch(primitive [], key)	This method is useful for searching a sorted array for a key. This returns an index of the value.
static int binarySearch(T[], key, Comparator)	This method is useful to search a Comparator sorted array for a given value.
static boolean equals(Object[], Object[]) static boolean equals(primitive[], primitive[])	This method is useful to check if two arrays contain equal content or not.
public static void sort(Object[]) public static void sort(primitive[])	This method is useful to sort the array on which it is used by natural order.
public static void sort(T[], Comparator)	This method takes Comparator to sort the elements of an array.
public static String toString(Object[]) public static String toString(primitive[])	This method prints the content of an Array in String form.

13.5.2 Collections (java.util.Collections)

Table 13.7 lists methods that are useful for Collections. These methods can be used to work on Collections.

Table 13.7 List of Collections methods

Method	Description
static int binarySearch(List, key) static int binarySearch(List, key, Comparator)	This method is similar to Arrays, where it searches a sorted list for a given key. It then returns an Index. The overloaded version of this accepts Comparator parameter to search and returns an index of the value.
static void reverse(List)	This method is useful to reverse the order of elements in a given List.
static Comparator reverseOder() static Comparator reverseOder(Comparator)	This method is useful to accept Comparator to reverse the current sort sequence and return this Comparator back.
static void sort(List) static void sort(List, Comparator)	This method is useful to sort the given list by natural order. The overloaded method accepts a Comparator to sort.

13.5.3 Key Methods for List, Set, Map, and Queue

Table 13.8 lists key methods you will encounter periodically while working with List, Set, Map, and Queue.

Table 13.8 List of key methods for List, Set, Map, and Queue

Method	List	Set	Map	Description
boolean add(element)	X	X		This method is useful to add an element to a Collection. In case of list, an overloaded method is available which allows to add an element at a specific index.
boolean add(index, element)	X			
boolean contains (object)	X	X		This method is useful for searching a collection for an object in order to find out if the object is present in the collection or not. This is done by returning a boolean value.
boolean containsKey(object key)			X	
boolean containsValue(object value)			X	For Maps, there are two overloaded versions of the method – one which looks for an object based on a key and the other which is based on a value.

Table 13.8 (Continued)

Method	List	Set	Map	Description
object get(index)	X			This method is useful to get an object for a given index. In case of Map, there is an overloaded method which accepts Key to get the object.
object get(key)			X	
int indexOf(object)	X			This method returns the location of the object in a List.
Iterator iterator()	X	X		This method returns iterator for a List or a Set. Please note that Map does not have this method.
Set keyset()			X	This method is useful for Map to return Set of Map's keys.
put(key, value)			X	This method is useful to add a key/value pair to a Map.
remove(index)	X			This method is useful to remove an element for a given index. There is an overloaded method for Set which accepts object as parameter to remove. Map has an overloaded method which accepts key to remove an element from Map for the specified key.
remove(object)	X	X		
remove(key)			X	
int size()	X	X	X	This method returns the number of elements in a collection.
object[] toArray() T[] toArray(T[])	X	X		This method returns a collection of elements in an array form.

The following are some of the operations that are possible on Lists in Java:

1. **`void add(int index, object o):`** As the name of the operation suggests, this method is used to insert elements into the list. It should, however, be noted that even when elements are inserted anywhere in between the list, there is no chance of overwriting since all other elements are shifted to make room for the newly added element in the list. The index in the parameter of the method indicates the location where the new element needs to be added to the list.
2. **`boolean addAll(int index, Collection c):`** All of the elements present in the Collection, c, being inserted into the parameters of the method will be added to the list for which the method is being invoked. Again, the elements that were already present in the list are shifted to accommodate the elements that are to be added, ensuring that elements are not overwritten. If the list for which the operation is being invoked changes, it will return true. Otherwise, the value will be false.
3. **`Object get(int index):`** This operation will return the object that is stored at the location of the index that is input as a parameter.
4. **`int indexOf(Object obj):`** This operation will return the first instance of the instantiation of the object in the list where it is being invoked. In case the object that was input as a parameter was not a part of the list that invoked it, .1 will be returned to the user as an indication that the object does not exist from where it was invoked.
5. **`int lastIndexOf(object obj):`** Similar to the operation mentioned in point 4, the `lastIndexOf( )` operation returns the last instance of the object of the list where it is being invoked. In case the object does not exist as an element of the list, .1 will be returned to the user as an indication that something is wrong.
6. **`ListIterator listIterator( ):`** Whenever this operation is called, an iterator is returned to the beginning of the list for which the operation was being invoked.

13.5.3.1 Comparable Interface

`Collections.sort()` uses the Comparable interface to sort Lists. Similarly, `java.utils.Arrays.sort()` uses it to sort arrays of objects. This interface requires the implementing classes to implement the `compareTo()` method. The following example will help you to understand this better.

```
package java11.fundamentals.chapter13;
public class Food implements Comparable{
    private String item;

    @Override
    public int compareTo(Object o) {
        if(o instanceof Food) {
            Food f = (Food) o;
            return item.compareTo(f.getItem());
        }
        return 0;
    }
    public String getItem() {
        return item;
    }
    public void setItem(String item) {
        this.item = item;
    }
}
```

The above class implements Comparable interface and implements `compareTo(Object o)` method to compare two food objects. However, with this plain implementation, we have to check if the incoming object is `instanceof Food` and then we perform `compareTo` on String. String implements `compareTo` internally so it is easier to just compare on String attributes of objects. However, we can improve this by using the Generics version as follows:

```
package java11.fundamentals.chapter13;
public class FoodWithGenerics implements Comparable<Food>{
    private String item;

    @Override
    public int compareTo(Food f) {
        return item.compareTo(f.getItem());
    }

    public String getItem() {
        return item;
    }
    public void setItem(String item) {
        this.item = item;
    }
}
```

In the above example, using Comparable with Generics reduces the code significantly. This also guarantees that the incoming object is definitely of Food type.

Table 13.9 shows the return value for `compareTo()`.

Table 13.9 Return value for `compareTo()`

Condition	Result
currentObject < compareToObject	negative
currentObject == compareToObject	zero
currentObject > compareToObject	positive

13.5.3.2 Comparator Interface

As we have seen earlier, there are two sort methods, one is the `Collections.sort()` that takes List, and the other one is the overloaded method that takes a List and Comparator. This interface gives the ability to sort a given collection in many different methods. Another benefit of this interface is that it allows sorting of any class even if we cannot modify it. This is contrary to Comparable, where we need to modify the class to implement the `compareTo()` method. This is a great advantage, as many times we want to sort objects based on multiple methods and we also do not have access to the source its code. Just like Comparable interface, the Comparator interface is also very easy to implement, which offers a single method `compare()`. The following code will help you to understand this better.

```
package java11.fundamentals.chapter13;
import java.util.ArrayList;
import java.util.List;
public class FoodWithComparator{
    private String item;

    public static void main(String args[]) {
        List<Food> junkFoodItems = new ArrayList<Food>();

        //Create a list of Food
        junkFoodItems.add(addItemToFoodList("Pizza"));
        junkFoodItems.add(addItemToFoodList("French Fries"));
        junkFoodItems.add(addItemToFoodList("Milk Shake"));
        junkFoodItems.add(addItemToFoodList("Burger"));
        junkFoodItems.add(addItemToFoodList("Fried Chicken"));

        System.out.println("Before Sorting : " + junkFoodItems.toString());

        //Create an object for Comparator to sort by item
        SortForComparator sfc = new SortForComparator();

        junkFoodItems.sort(sfc);

        System.out.println("After Sorting : " + junkFoodItems.toString());

    }

    public static Food addItemToFoodList(String item) {
        Food f = new Food();
        f.setItem(item);
        return f;
    }

    public String getItem() {
        return item;
    }
    public void setItem(String item) {
        this.item = item;
    }
}
```

The above class uses a custom comparator that we have created as shown below. Also note that we are using the Food class which we have created in the earlier in Comparable example in Section 13.5.3.1.

```
package java11.fundamentals.chapter13;
import java.util.Comparator;
public class SortForComparator implements Comparator<Food>{
    @Override
    public int compare(Food f1, Food f2) {
        return f1.getItem().compareTo(f2.getItem());
    }
}
```

As you can see, we use food item to sort. This method is similar to the `compareTo()` method. In fact, in that method, we used the `compareTo()` method from String implementation to compare two strings. Executing the above program gives the following result.

```
Before Sorting : [java11.fundamentals.chapter13.Food@7a79be86, java11.fundamentals.chapter13.Food@34ce8af7, java11.fundamentals.chapter13.Food@b684286,
java11.fundamentals.chapter13.Food@880ec60, java11.fundamentals.chapter13.Food@3f3afe78]
After Sorting : [java11.fundamentals.chapter13.Food@880ec60, java11.fundamentals.chapter13.Food@34ce8af7, java11.fundamentals.chapter13.Food@3f3afe78,
java11.fundamentals.chapter13.Food@b684286, java11.fundamentals.chapter13.Food@7a79be86]
```

The program produces an interesting output. Although the program has sorted the objects based on the items' natural order, we are not able to see them. Can you guess why we see this instead of Food item names? This is because we have not overridden the `toString()` method and we are using it from the base class Object, which only prints Class name followed by @ and hashcode. You can easily fix this. For a reminder on how to do so, go to the Section 13.2.4.1 "Overriding `toString()` Method".

The following is the updated Food Class with overridden `toString()` method.

```
package java11.fundamentals.chapter13;
public class FoodWithToString implements Comparable{
    private String item;

    @Override
    public int compareTo(Object o) {
        if(o instanceof FoodWithToString) {
            FoodWithToString f = (FoodWithToString) o;
            return item.compareTo(f.getItem());
        }
        return 0;
    }
    public String getItem() {
        return item;
    }
    public void setItem(String item) {
        this.item = item;
    }


    public String toString() {
         return this.item;
    }

}
```

Now, we need to update the Comparator implemented class and program, which is given below.

```
package java11.fundamentals.chapter13;
import java.util.Comparator;
public class SortForComparatorWithFoodWithToString implements
Comparator<FoodWithToString>{
    @Override
    public int compare(FoodWithToString f1, FoodWithToString f2) {
        return f1.getItem().compareTo(f2.getItem());
    }
}
```

And now the program can use this newly created SortForComparatorWithFoodWithToString class for sorting.

```
package java11.fundamentals.chapter13;
import java.util.ArrayList;
import java.util.List;
public class FoodWithComparatorWithFoodToString{
    private String item;

    public static void main(String args[]) {
        List<FoodWithToString> junkFoodItems = new ArrayList<FoodWithToString>();

        //Create a list of FoodWithToString
        junkFoodItems.add(addItemToFoodList("Pizza"));
        junkFoodItems.add(addItemToFoodList("French Fries"));
        junkFoodItems.add(addItemToFoodList("Milk Shake"));
        junkFoodItems.add(addItemToFoodList("Burger"));
        junkFoodItems.add(addItemToFoodList("Fried Chicken"));

        System.out.println("Before Sorting : " + junkFoodItems.toString());

        //Create an object for Comparator to sort by item
        SortForComparatorWithFoodWithToString sfc = new
SortForComparatorWithFoodWithToString();

        junkFoodItems.sort(sfc);

        System.out.println("After Sorting : " + junkFoodItems.toString());

    }

    public static FoodWithToString addItemToFoodList(String item) {
        FoodWithToString f = new FoodWithToString();
        f.setItem(item);
        return f;
    }

    public String getItem() {
        return item;
    }
    public void setItem(String item) {
        this.item = item;
    }
}
```

Let us run this program and analyze the output.

```
Before Sorting : [Pizza, French Fries, Milk Shake, Burger, Fried Chicken]
After Sorting : [Burger, French Fries, Fried Chicken, Milk Shake, Pizza]
```

From the above program, you can see how easy it is to use Comparator to sort collections.

13.5.3.3 Comparable versus Comparator

Table 13.10 shows the difference between these the interfaces Comparable and Comparator.

Table 13.10 Difference between Comparable and Comparator

Comparable	Comparator
int firstObject.compareTo(secondObject)	int compare(firstObject, secondObject)
Returns the following: firstObject < secondObject – Negative firstObject > secondObject – Positive firstObject == secondObject – Zero	Returns similar values as Comparable
The Class that needs a sorting must implement Comparable. Hence, it needs to be modified.	The Class that needs a sorting need not be modified. Comparator allows building a separate class to sort desired elements. This class can later be used to sort the elements in any class.
With this, only One sort sequence is possible.	Since we are creating a separate class to sort, we can create as many classes as we want. Hence, we can create many sort sequences.
This is implemented by Java's core classes like String, Wrapper, Date, Calendar, etc.	This is intended to be used by third-party classes to sort instances.

Summary

Java improves the facilities of Generics and collections that were present in previous versions. It significantly enhances the capabilities of the classes that are available for creating customized set of objects. Programmers need to learn the concept of generic programming.

Generics present in Java provide the facility required for efficient programming. Java offers internal Generics as well as generic methods that allow programmers to improve their coding performance.

Java also provides a detailed collections framework. This framework is designed to allow programmers to create specific collections of objects and pass them around when using different Java APIs and other functional elements. The framework provides various interfaces and describes how these interfaces are often introduced by the collection classes, which are available for implementation in the programming language. We have described these collection classes, which you can use in a variety of applications.

There are various implementations of List, Map, Set, and Queue that all have their advantages when employed for keeping a record of the objects.

We also discussed stream API, which allows for an alternate way of creating useful connections and for using them to get the job done in a variety of environments.

In this chapter, we have learned the following concepts:

1. Generic programming and how to use it in a program.
2. Generic methods and uses.
3. Collections in Java and their benefits.
4. Collection interface and implementation of collection classes.

In Chapter 14, we will explore error handling, logical errors, semantic errors, try-catch-finally block, and checked versus runtime exceptions.

Multiple-Choice Questions

1. Which of the following is the most apt reason for using generics?
   - **(a)** Generics makes the code faster.
   - **(b)** Generics are useful for adding stability to the code by making bugs detectable during runtime.
   - **(c)** Generics are useful for making the code more readable and optimized.
   - **(d)** Generics are useful for adding stability to the code by making bugs detectable at compile time.
2. Which of the given parameters is utilized for a generic class to return and accept a number?
   - **(a)** V
   - **(b)** N
   - **(c)** T
   - **(d)** K
3. _____ permits us to invoke a generic method as a normal method.
   - **(a)** Inner Class
   - **(b)** Type Interface
   - **(c)** Interface
   - **(d)** All of the above
4. _____ consists of all the collection classes.
   - **(a)** Java.awt
   - **(b)** Java.util
   - **(c)** Java.net
   - **(d)** Java.lang
5. What do you understand by a Collection in Java?
   - **(a)** A group of classes
   - **(b)** A group of interfaces
   - **(c)** A group of objects
   - **(d)** None of the above

Review Questions

1. How are Generics useful?
2. What are the different collection interfaces?
3. Which collection class is used to store key–value pair data?
4. What is the difference between ArrayList and Vector?
5. Explain stream API.
6. What is LinkedList? How it is useful?
7. What are implementing classes of Map interface?
8. When is Set useful? How do we use it?

Exercises

1. Create a chart to show all the collection interfaces and implementation classes.
2. Write a program that demonstrates the use of all the collection classes.
3. Use a real life example to write a program to implement PriorityQueue.

Project Idea

Create a program to add and display non-persistent data of all the vehicles entered into a parking lot. Non-persistent data means that we cannot use the database to store this data and hence we have to hold this data in memory. The data should be captured in a way that we can segregate vehicle-specific and driver-specific information separately but linked to each other. Finally, display this information on a web page and the admin should be able to search the data based on various features such as color of the car and registration number.

Hint: Use various collection classes to hold this data in memory.

Recommended Readings

1. Philip Wadler, Maurice Naftalin. 2010. *Java Generics and Collections: Speed up the Java Development Process*. O'Reilly Media: Massachusetts
2. The Java™ Tutorials, Oracle: https://docs.oracle.com/javase/tutorial/java/generics/index.html
3. The Java™ Tutorials, Oracle: https://docs.oracle.com/javase/tutorial/collections/intro/index.html